CMSC426: Image Processing

Prof. Yiannis Aloimonos
Nitin J. Sanket and Kiran Yakkala
April 3

Benny Cheng

Project 3: Rotobrush

Local Windows

The SnapCut algorithm implements the idea of local windows to measure mappings of color and shape to determine the general location of the foreground that we want to track. To setup the local window, I first created a simple mask around the foreground of frame 1 via roipoly.

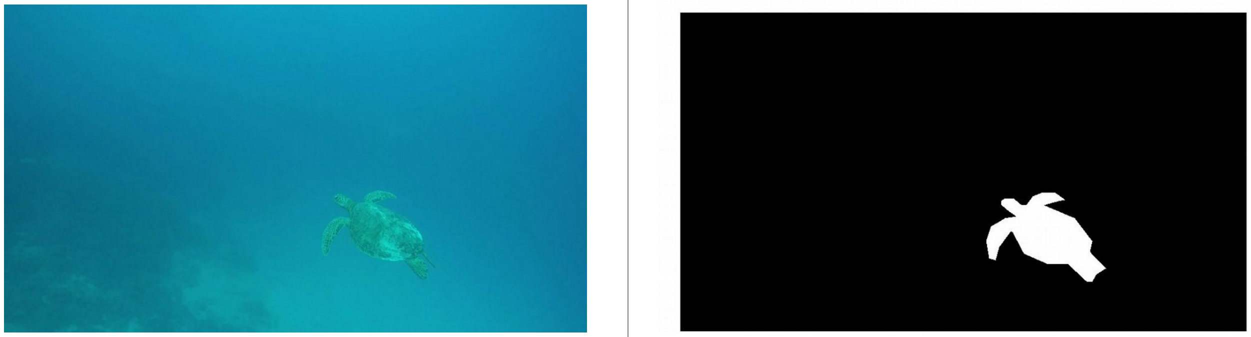


Figure 1: Roipoly around frame 1.

Having a rough estimate mask is sufficient for the rotobrush algorithm as we will plot windows of size N x N around the borders of the mask. These windows will be used to capture local foregrounds and background. It is key that these local windows are overlapping (roughly 1/3 of the window should coexist in its neighbors) such that future window propagations from frame to frame encapsulates the entirety of the mask. Note here that the size of N will affect computation speed, but will also produce safer estimates later on in the pipeline. I found that 40 is a good size to use for accuracy, and 30 speed. For my system, I used local windows of size 30 x 30 spread across at the border of the mask at intervals of half the window size. The result is depicted in Figure 2.

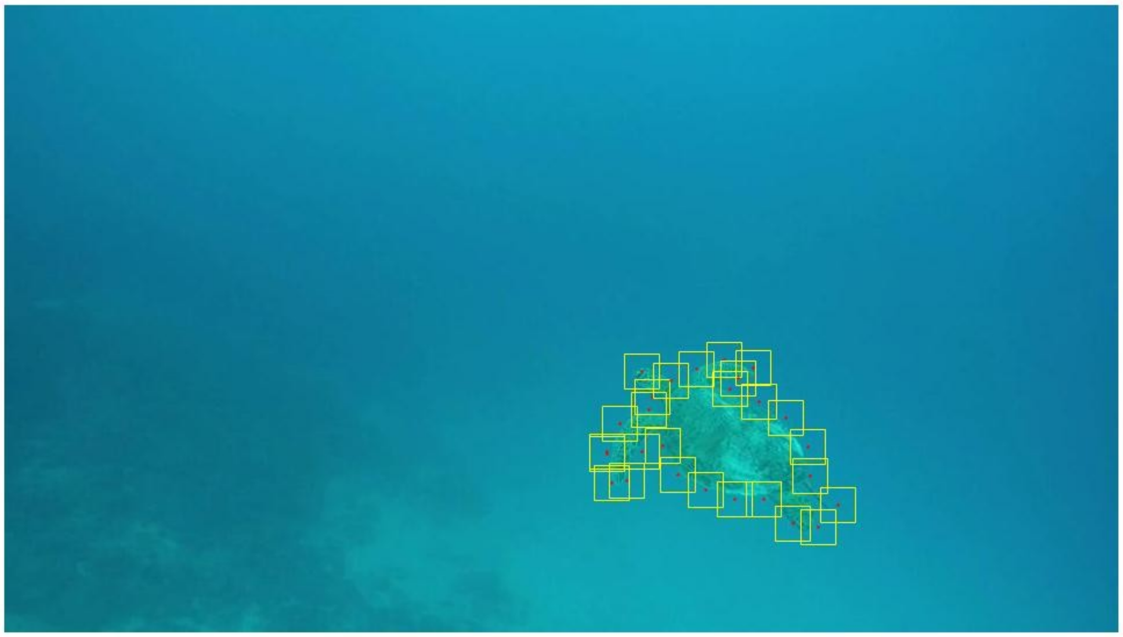


Figure 2: Initial position of local windows.

Foreground/Background/Boundaries

Before initializing color models for all the local windows, I first made a mask of the foreground pixels and background pixels based off of the roipoly mask created initially. Then I get the boundary of each window via bwboundaries. These will contribute to the calculation of the color models and shape models further down the road.

Initial Color Model

The color model will be used to distinguish pixels as foreground and background pixels based off of their color. Each window will have its own unique color model. To create a color model for a window:

1. Get the RGB image’s L*a*b information.
2. Train a Gaussian Mixture Model for all pixels of the window that is associated with the foreground. This is done through Matlab’s fitgmdist class.
3. Train a GMM for all pixels of the window that is the background.
4. Combine the output of the two models using the equation:

$$p_c(x) = p_c(x|\mathcal{F}) / \left(p_c(x|\mathcal{F}) + p_c(x|\mathcal{B})\right) ,$$

Where Pc (x|F) and Pc (x|B) are the probabilities given by the GMMs above.

Initial Color Model Confidence

The color model confidence will be used to entail how separable the foreground is from the background. Computing it entails just following equation 2 of the Snapcut paper in section 2.1. The sigma in the weighing function was fixed to be half my window size as well.

Initial Shape Model and Confidence

The shape model entails simply the foreground and background mask created in the beginning of our setup. The shape confidence mask is defined by equations 3 and 4 of the Snapcut paper (in sections 2.1 and 2.4, respectively).

Estimating Motion

To propagate our models to the next frame, we need to first find a mask for the second frame and recenter our current windows such that it is within relative proximity to its edges. To do so we first estimate the motion of the object as a whole by estimating its geometric transformation from frame 1 to frame 2. It is key here that we only sample points within the foreground of the image to align the foreground of our current frame to the next. Otherwise I found that the match will try to align the entire frame such that even the backgrounds are matching. As we only care about the foreground that should be avoided Next, apply an affine transformation between frame 1 and call it frame 2'. After having created frame 2', we will track the boundary movement between frame 1 and frame 2' by using Matlab's opticalFlowFarneback class to estimate the flow of the object from frame 1 to frame 2'. Within each window, calculate the average flow vectors and readjust the center by that vector. This will give us a rough estimate of where our new windows will be located in frame 2. Figure 3 depicts the updated window locations from frame 1 to frame 2. Using optical flow to recenter may cause shifts in where the windows are centered. The overlapping windows were created such that even after re-centering, our windows will still cover the entirety of the foreground.

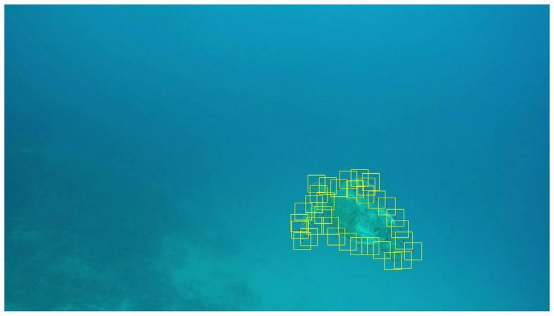
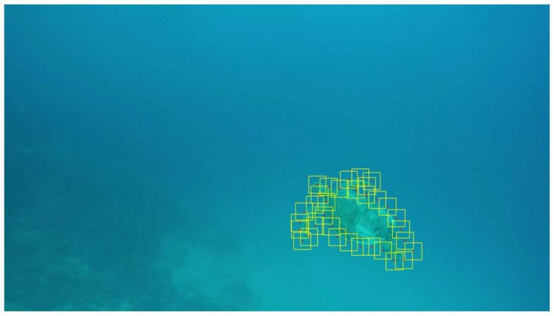
Frame 1	Frame 2
	

Figure 3: Updated window locations from frame 1 to frame 2.

Updating

Update Color and Shape Models

Now that we have new local windows for the next frame, we need to update the color and shape models to reflect that change. The affine transformed image is sufficient for our updated shape model as it quite accurately depicts the boundaries of the next frame. These can be carried over from the mask of frame 2'. Updating the color model will be more complex. We first create another color model by retraining GMM models for the foreground and background of this current frame and the previous. This time, instead of sampling from a distance from the boundary, we sample pixels above a certain threshold. On my system, for the foreground we sample above 0.75 and for the background we sample above 0.2. Take this color model, call it Gt+1, and call the former color model Gt. If Gt+1 has fewer pixels in the foreground we choose to keep the new color model otherwise we stick with Gt. This is done because we assume that since we are tracking the object by its foreground boundaries. Therefore it is common for the foreground object to experience minimal change whereas the background color can change drastically. Therefore, if we notice that the foreground pixels suddenly changed drastically, we know that we cannot trust the new color model. If we choose to update the color model, we must then update the color confidence value as well.

Combining Shape and Color Models

After having the update shape and color models, we can simply combine them via the equation:

$$p_{\mathcal{F}}^{k}(x) = f_s(x)L^{t+1}(x) + (1 - f_s(x))p_c(x).$$

Where P^k_f (x) is the foreground map for the kth window, F_s(x) is the shape confidence map, L_{t+1} (x) is the shape model's foreground mask, and p_c (x) is the color model's foreground mask.

Merge and Extract Final Mask

Now having combined all the models of all the windows, we now have a foreground probability map for each local window. We now merge them all together to get a global foreground map via:

$$p_{\mathcal{F}}(x) = \frac{\sum_k p_{\mathcal{F}}^{k}(x)(|x - c_k| + \epsilon)^{-1}}{\sum_k(|x - c_k| + \epsilon)^{-1}},$$

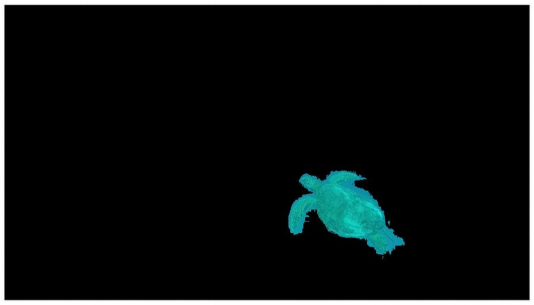
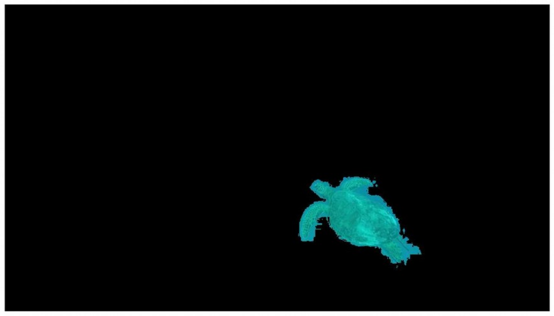
- k - index of local windows (the sum ranges over all the k-s such that the updated window W k t+1 covers the pixel)
- epsilon - a small constant (0.1 in the system),
- c_k - the center of the window (|x − c k | is the distance from the pixel x to the center).

To implement this, I take each foreground mask from each local window and plot them with respect to where it's window co-ordinates are. Sum those values up with respect to the equation above and fill the value inside. This however, returns a real-valued probability mask but we want a binary mask. To fix this I simply take a threshold (0.9 in my system) to be considered the foreground. In theory, a lazy snapping algorithm might be better.

The updating phase is then repeated for all subsequent frames, utilizing elements from the previous windows to determine new GMM models as well as masks.

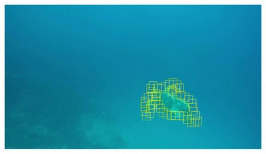
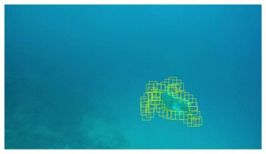
Iterative Refinement

I wasn't satisfied with the rough edges so I decided to iterate the steps above to produce a more accurate boundary. To do this, I simply updated the foreground map and recalculated the models and reapplied it to the current frame. The documentation states that the process usually converged about the 3rd-4th iteration so I repeated the step 3 times. The product looked promising at some windows.

 Without iteration	 With iteration

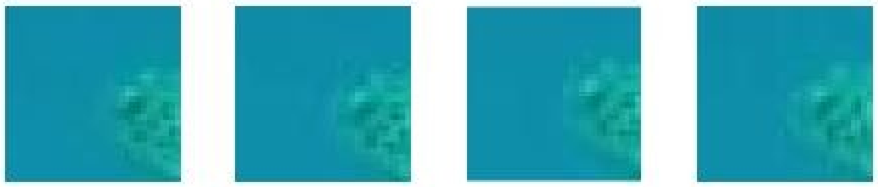
Conclusion

Window Locations

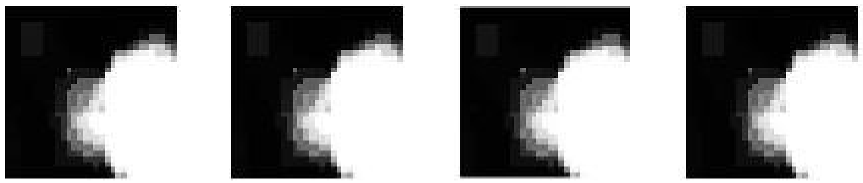
Frame 1	Frame 2	 Frame 3	 Frame 4

Frame 1 Frame 2 Frame 3 Frame 4

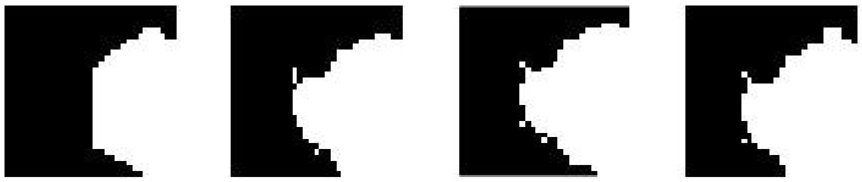
Image



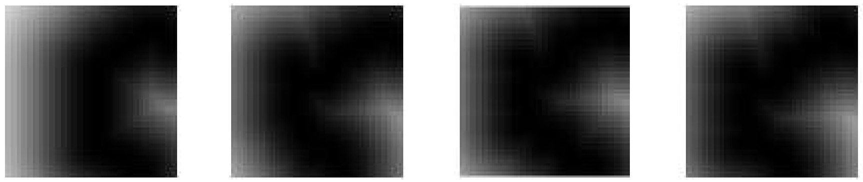
Color Model



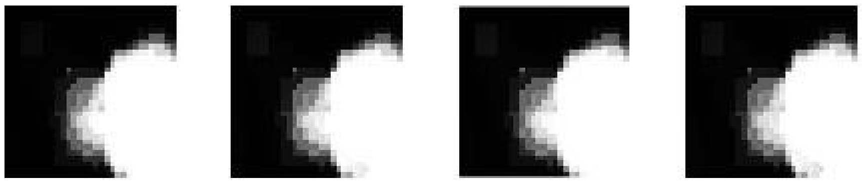
Shape Model



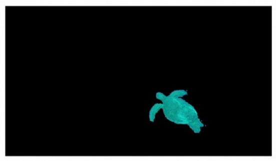
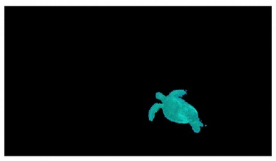
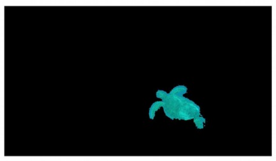
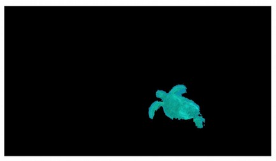
Shape Confidence

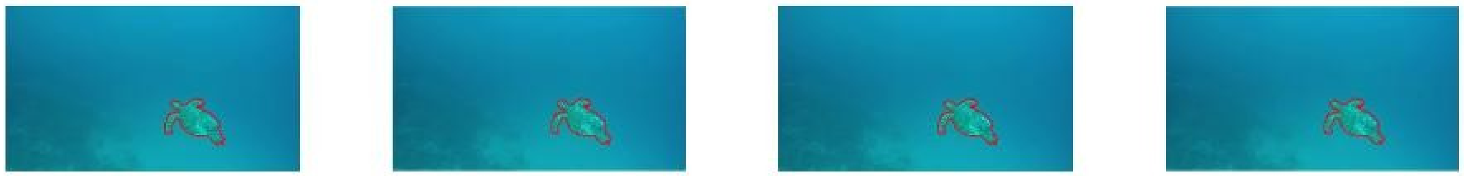


Shape and Color Model Merge



Final Object Boundary

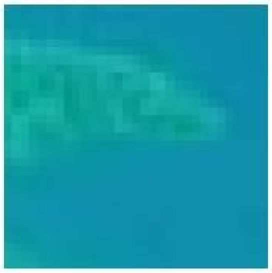


Discussion

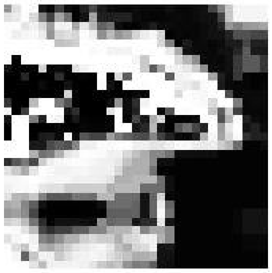
Throughout the implementation of the rotoscope feature, I’ve noted a few things:

- Larger window size account for inaccurate optical flows, but computation speed decreases as you are accounting for more pixels per window.
- Image with more distinguishable background and foreground segmentations account for cleaner foreground masks. With the sea turtle there are some spots where the GMM can’t distinguish the foreground from the background well enough and as a result lead to artifacts along the edge. This information thus is carried onto the next frame and the frames after that leading to increasingly false color models.

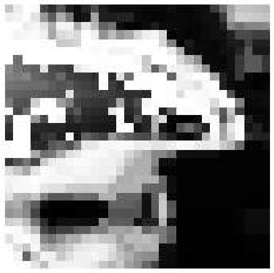
Image



Color Model



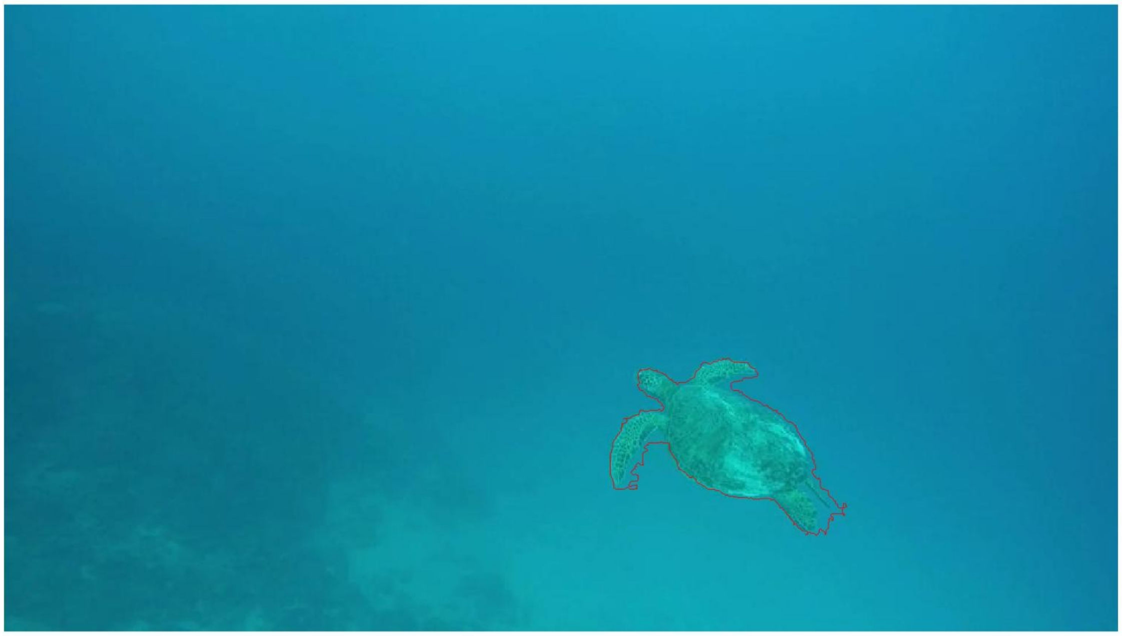
Foreground Boundary



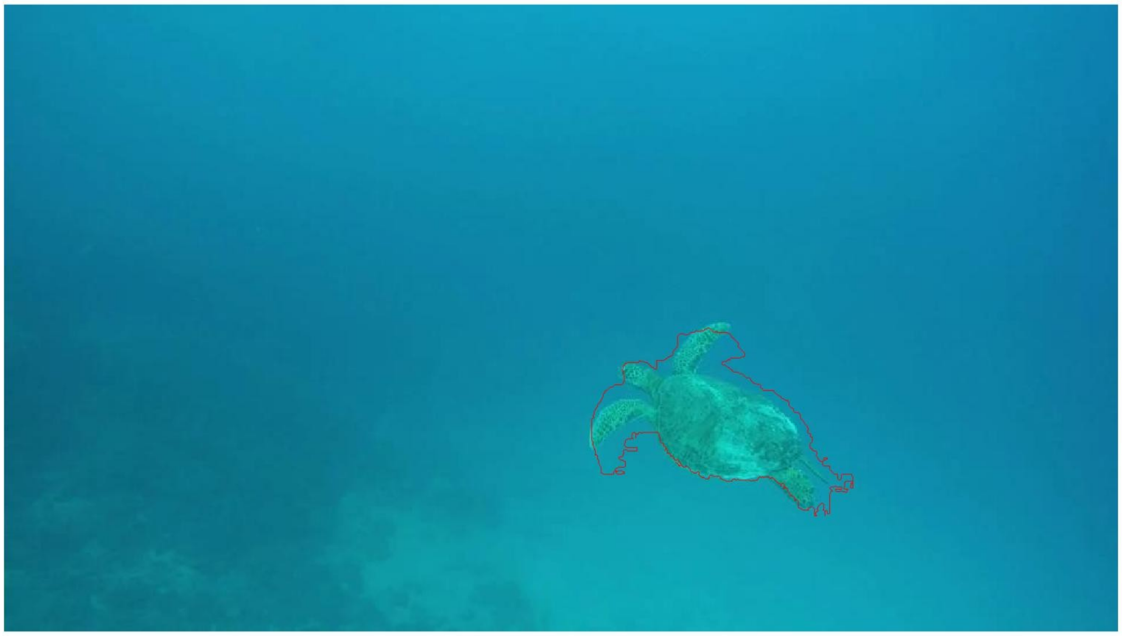
While initial frames and a few subsequent frames might look good, as the error grows, we start to see more false boundaries as a result. This comes from errors within the system such as false color models and inaccurate optical flows. Note the rather tight boundary around the turtle in the first image (Frame 1) below compared to the rough boundary of the last image (Frame 2).

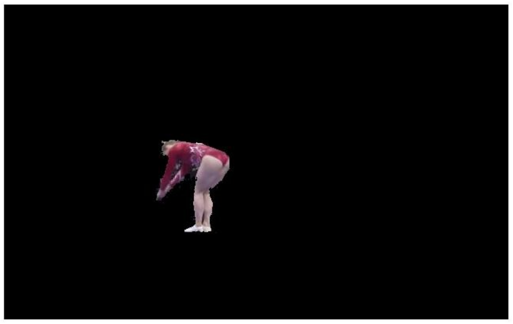
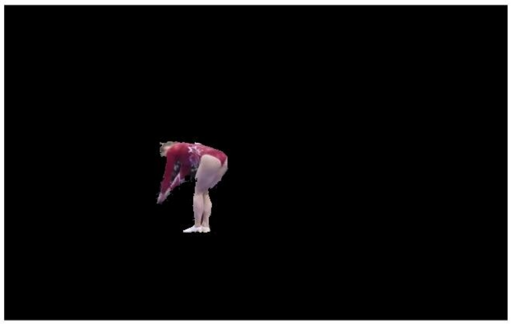
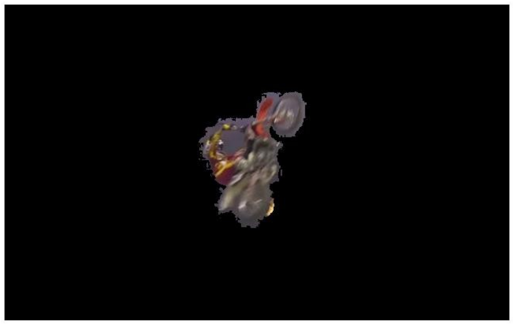
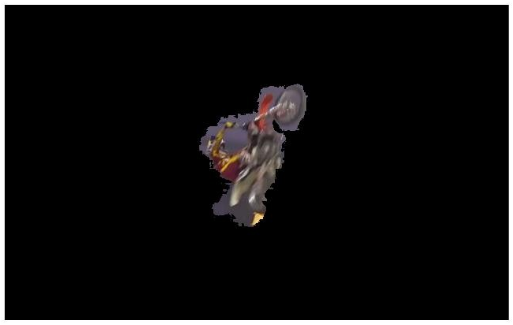
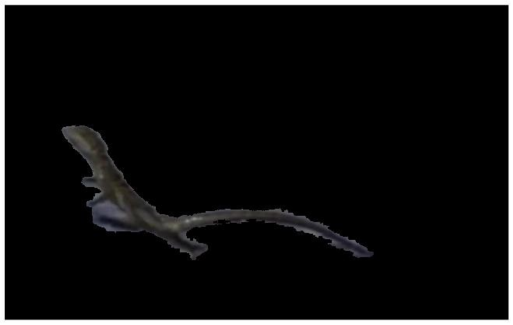
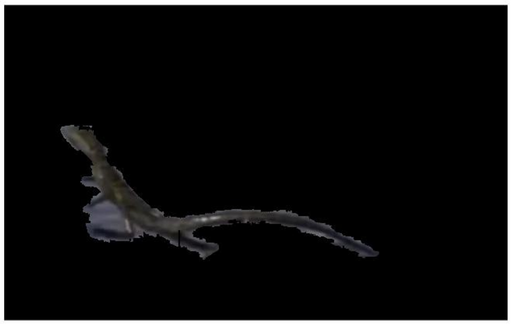
Higher resolution example:

Frame 1



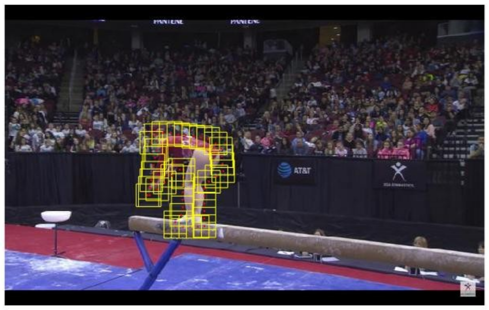
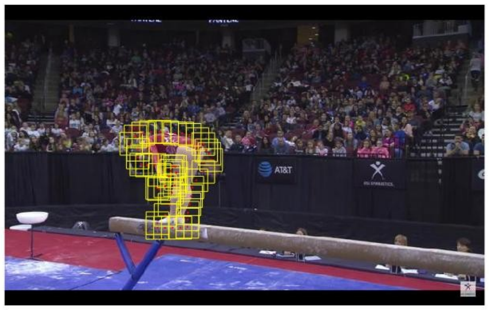
Frame 10



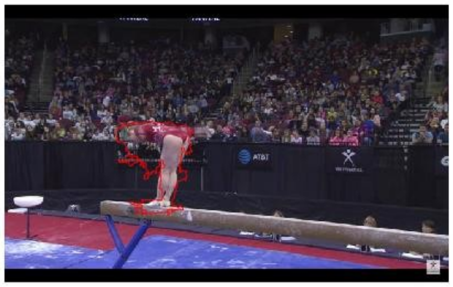
	
	
	

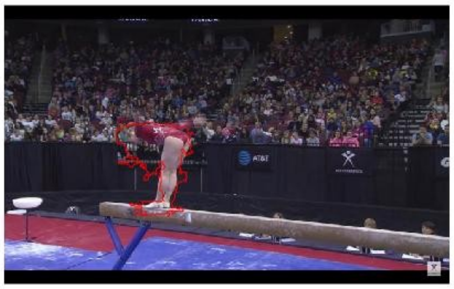
Notice how the gymnast's bright red outfit pose a stark contrast to the foreground as compared to the biker's bike and arms and the lizard's body that almost blend into the background. And notice how much cleaner the segmentation is of the gymnast as compared to the other 2 images between frames.

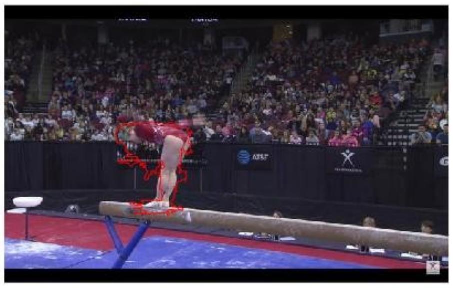
- Spontaneous movements and cross sections make for bad boundary detection.

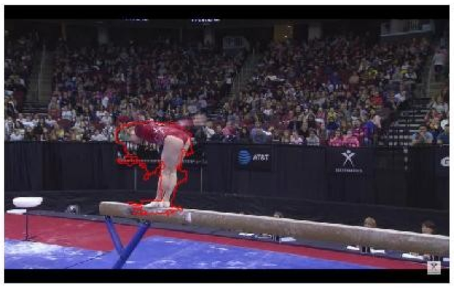
 Frames3: Frame 37	 Frames3: Frame 46

Since the initial boundary was drawn to accommodate two hands in parallel, when they split apart at frame 46 as the gymnast is preparing to flip, we lose significant attention to many of the actual foreground boundary. As the hand starts to cross behind her, tracking points also lose the sense of flow and those points remain stagnant for the rest of the iteration. As a result, the first few frames of the

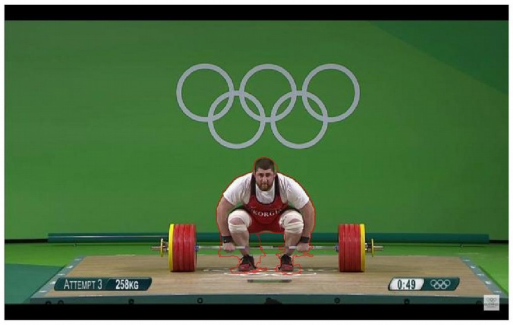
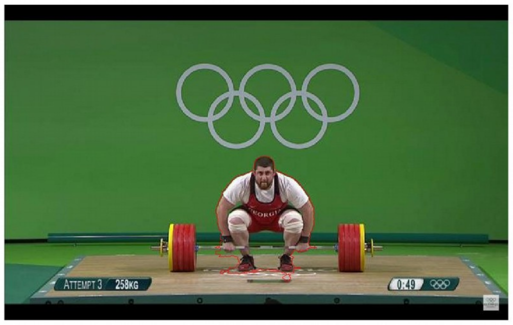








video, where minimal transitions were occurring, looked clean, whereas once big movements starts to happen we start losing detail on the foreground mask. The figure below demonstrates frames 40-44.

 Frames4: frame 1	 Frames4: frame 10

With minimal dynamic changes to the boundary of the foreground, and with more separable color models we get much cleaner segmentation throughout, as depicted by the weight lifter above.

- The documentation also defines manual adjustment options.
  - Might implement later
- Matlab's lazysnapping algorithm provides a rough estimation for boundaries that are a lot smoother than just thresholding. But since the process requires breaking the image into label matrix, it requires more computation for better estimations, and at lower computations completely removes some foregrounds.
- Since the full implementation is not being designed, errors will occur during feature matching leading to lost of foreground tracking ability (since we only match on points depicted within the foreground). This can occur either through spontaneous and rapid movements or shrinking foreground masks such that SURF doesn't correctly detect matching points. The output videos reflect the span at which the foreground object is still able to be detected by the algorithm.
- Videos are in AVI format as linux does not support MPEG-4.